

Matching Multiplications in Bit-Vector Formulas

Supratik Chakraborty¹, Ashutosh Gupta², and Rahul Jain²

¹ Indian Institute of Technology Bombay, India

² Tata Institute of Fundamental Research, India

Abstract. Bit-vector formulas arising from hardware verification problems often contain word-level arithmetic operations. Empirical evidence shows that state-of-the-art SMT solvers are not very efficient at reasoning about bit-vector formulas with multiplication. This is particularly true when multiplication operators are decomposed and represented in alternative ways in the formula. We present a pre-processing heuristic that identifies certain types of decomposed multipliers, and adds special assertions to the input formula that encode the equivalence of sub-terms and word-level multiplication terms. The pre-processed formulas are then solved using an SMT solver. Our experiments with three SMT solvers show that our heuristic allows several formulas to be solved quickly, while the same formulas time out without the pre-processing step.

1 Introduction

In recent years, SMT solving has emerged as a powerful technique for testing, analysis and verification of hardware and software systems. A wide variety of tools today use SMT solvers as part of their core reasoning engines [1,2,3,4,5,6,7,8,9,10]. A common approach used in several of these tools is to model the behaviour of a system using formulas in a combination of first-order theories, and reduce the given problem to checking the (un)satisfiability of a formula in the combined theory. SMT solvers play a central role in this approach, since they combine decision procedures of individual first-order theories to check the satisfiability of a formula in the combined theory. Not surprisingly, heuristic techniques to improve the performance of SMT solvers have attracted significant attention over the years (see [11,12] for excellent expositions). In this paper, we add to the repertoire of such heuristics by proposing a pre-processing step that analyzes an input formula, and adds specially constructed assertions to it, without changing the semantics. We focus on formulas in the quantifier-free theory of fixed-width bit-vectors with multiplication, and show by means of experiments that three state-of-the-art SMT solvers benefit significantly from our heuristic when solving many benchmarks with multiplication operators.

The primary motivation for our work comes from word-level bounded model checking (WBMC) [4,1] and word-level symbolic trajectory evaluation (WSTE) [13] of embedded hardware systems. Specifically, we focus on systems that process data, represented as fixed-width bit-vectors, using arithmetic operators. When reasoning about such systems, it is often necessary to check whether

a high-level property, specified using bit-vector arithmetic operators (viz. addition, multiplication, division), is satisfied by a model of the system implementing a data-processing algorithm. For reasons related to performance, power, area, ease of design etc., complex arithmetic operators with large bit-widths are often implemented by composing several smaller, simpler and well-characterized blocks. For example, a 128-bit multiplier may be implemented using one of several multiplication algorithms [14,15,16] after partitioning its 128-bit operands into narrower blocks. SMT formulas resulting from WBMC/WSTE of such systems are therefore likely to contain terms with higher-level arithmetic operators (viz. 128-bit multiplication) encoding the specification, and terms that encode a lower-level implementation of these operators in the system (viz. a Wallace-tree multiplier). Efficiently reasoning about such formulas requires exploiting the semantic equivalence of these alternative representations of arithmetic operators. Unfortunately, our study, which focuses on systems using the multiplication operator, reveals that three state-of-the-art SMT solvers (Z3 [17], CVC4 [18] and BOOLECTOR [2]) encounter serious performance bottlenecks in identifying these equivalences. Our limited experiments show that these bottlenecks manifest most conspicuously when reasoning about the unsatisfiability of formulas.

A motivating example: To illustrate the severity of the problem, we consider the SMT formula arising out of WSTE applied to a pipelined serial multiplier circuit, originally used as a benchmark in [13]. The circuit reads in two 32-bit operands sequentially from a single 32-bit input port, multiplies them and makes the 64-bit result available in an output register.

The property to be checked asserts that if a and b denote the word-level operands that are read in, then after the computation is over, the output register indeed has the product $a *_{[32]} b$, where $*_{[32]}$ denotes 32-bit multiplication. The system implementation, as used in [13], is described in SYSTEMVERILOG (a hardware description language) and makes use of the multiplication operator (i.e., $*$) in SYSTEMVERILOG with 32-bit operands. The Language Reference Manual of SYSTEMVERILOG specifies that this amounts to using a 32-bit multiplication operation directly. The SMT formula resulting from a WSTE run on this example therefore contains terms with only 32-bit multiplication operators, and no terms encoding a lower-level multiplier implementation. This formula is proved unsatisfiable in a fraction of a second by BOOLECTOR, CVC4 and Z3.

We now change the design above to reflect the implementation of 32-bit multiplication by the long-multiplication algorithm [14], where each 32-bit operand is partitioned into 8-bit blocks. The corresponding WSTE run yields an SMT formula that contains terms with 32-bit multiplication operator (derived from the property being checked), and also terms that encode the implementation of a 32-bit multiplier using long-multiplication. Surprisingly, none of BOOLECTOR, CVC4 and Z3 succeeded in deciding the satisfiability of the resulting formula even after 24 hours on the same computing platform as in the original experiment. The heuristic strategies in these solvers failed to identify the equivalence of terms encoding alternative representations of 32-bit multiplication, and proceeded to bit-blast the formulas, resulting in this dramatic blowup in run-time.

Problem formulation: The above example demonstrates that the inability to identify semantic equivalence of alternative representations of arithmetic operators plagues multiple state-of-the-art SMT solvers. This motivates us to ask: *Can we heuristically pre-process an SMT formula containing terms encoding alternative representations of bit-vector arithmetic operators, in a solver-independent manner, so that multiple solvers benefit from it?* We answer this question positively in this paper, for the multiplication operator. The motivating example, that originally timed out after 24 hours on three solvers, is shown to be unsatisfiable by Z3 in 0.073s and by CVC4 in 0.017s, after applying our heuristic. Although BOOLECTOR does not benefit from our heuristic on this example, it benefits in several other examples, as discussed in Section 5.

Term re-writing vs adding tautological assertions: Prima facie, the above problem can be solved by reverse-engineering a lower-level representation of a bit-vector arithmetic operator, and by re-writing terms encoding this representation with terms using the higher-level bit-vector operator. Indeed, variants of this approach have been used earlier in different contexts [19,20,21,22,23]. In the context of SMT solving, however, more caution is needed. As shown in Example 2 of Section 2.2, the same collection of terms (in this case, sums-of-partial-products) can arise from two different word-level multiplication operations. This makes it difficult to decide which of several term re-writes should be used when there are alternatives. Even if the above dilemma doesn’t arise, re-writing one term with another is a “peep-hole” transformation that may not always correlate with improved solver performance. For example, one term may enable a re-write rule that helps simplify one sub-formula, while a syntactically distinct but semantically equivalent term may enable another re-write rule that helps simplify another sub-formula. Re-writing one term by another prevents both terms from jointly contributing to simplifications and improving the solver’s performance.

In this paper, we propose a heuristic alternative to term re-writing when solving bit-vector formulas with multiplication. Given a bit-vector formula φ containing terms with different representations of multiplication, our heuristic searches for patterns in the terms corresponding to two multiplication algorithms, i.e., long multiplication and Wallace-tree multiplication. Instead of re-writing the matched terms directly with bit-vector multiplication terms, we conjoin φ with assertions that semantically equate a matched term with the corresponding bit-vector multiplication term. Note that each added assertion is a tautology, and hence does not change the semantics of the formula. Since no re-writes are done, we can express multiple semantic equivalences without removing any syntactic term from the formula. This is an important departure from earlier techniques, such as [21], that rely on sophisticated re-writes of the formula. Our experiments show that the added tautological assertions succeed in preventing bit-blasting while solving in several cases, while in other cases, they help in pruning the search space even after bit-blasting. Both effects eventually translate to improved performance of the SMT solver. Furthermore, since our heuristic simply adds assertions to the input formula, it is relatively independent of the internals of any specific solver, and can be used with multiple solvers.

2 Preliminaries

In this section, we present some basics of the theory of quantifier-free fixed-width bit-vector formulas (**QF_BV**), and discuss two well-known multiplication algorithms of interest.

2.1 QF_BV: A short introduction

A bit-vector is a fixed sequence of bits. We denote bit-vectors by x, y, z etc. and often refer to blocks of bits in a bit-vector. For example, we may declare that a bit-vector x is accessed in blocks of width w . Let x_i denote the i th block of bits, with the block containing the least significant bit (LSB) having index 1.

A **QF_BV** term t and formula F are constructed using the following grammar

$$\begin{aligned} t &::= t * t \mid t + t \mid x \mid n^w \mid t \bullet t \dots \\ F &::= t = t \mid t \bowtie t \mid \neg F \mid F \vee F \mid F \wedge F \mid F \oplus F \mid \dots \end{aligned}$$

where x is a bit-vector variable, n^w is a binary constant represented using w bits, \bowtie is a predicate in $\{\leq, <, \geq, >\}$, and \bullet is a binary operator that concatenates bit-vectors. Note that we have only presented those parts of the **QF_BV** grammar that are relevant to our discussion. For more details, the reader is referred to [24,11]. We assume that all variables and arithmetic operators are unsigned. Following the SMT-LIB [25] convention, we also assume that arguments and results of an arithmetic operator have the same bit width. Let $\text{len}(t)$ denote the bit width of a term t . If $w \geq \text{len}(t)$, let $\text{zeroExt}(t, w)$ be a shorthand for $0^{w-\text{len}(t)} \bullet t$.

If an operator op is commutative, when matching patterns, we will not make a distinction between $a \text{ op } b$ and $b \text{ op } a$. We use the notation “ $t = s$ ” to denote that terms t and s are *syntactically identical*. The usual equality predicate, i.e. “ $=$ ”, is used to denote *semantic equivalence*. Given bit-vector terms x, y , and t , suppose $w = \max(\text{len}(x) + \text{len}(y), \text{len}(t))$. We use “ $[x * y = t]$ ” to denote the term $x' * y' = t'$, where $x' = \text{zeroExt}(x, w)$, $y' = \text{zeroExt}(y, w)$, and $t' = \text{zeroExt}(t, w)$. Similarly, the notation $[x * y]$ is used to denote $x' * y'$, where $x' = \text{zeroExt}(x, \text{len}(x) + \text{len}(y))$ and $y' = \text{zeroExt}(y, \text{len}(x) + \text{len}(y))$.

State-of-the-art SMT solvers for **QF_BV** apply several theory-specific simplification and re-write passes to decide the satisfiability of an input **QF_BV** formula. If the application of these passes does not succeed in solving the problem, the solvers eventually bit-blast the formula, i.e., translate it to an equivalent propositional formula on the constituent bits of the bit-vectors. This reduces the bit-vector satisfiability problem to one of propositional satisfiability (SAT). The bit-blasted problem is then solved using conflict driven clause learning (CDCL)[26,27] based SAT decision procedures. Among the leading SMT solvers for **QF_BV** available today are Z3[17], BOOLECTOR[2] and CVC4[18]; we use these extensively in our experiments to empirically evaluate our heuristic.

In the subsequent discussion, we assume access to a generic **QF_BV** SMT solver, called SMTSOLVER, with a standard interface. We assume that the interface provides access to two functions: (a) $\text{add}(F)$, that adds a formula F to the context

of the solver, and (b) *checkSat()*, that checks the satisfiability of the conjunction of all formulas added to the context of the solver. Note that such interfaces are commonly available with state-of-the-art SMT solvers, viz. BOOLECTOR, CVC4 and Z3.

2.2 Multipliers

As discussed in Section 1, there are several alternative multiplier implementations that are used in hardware embedded systems. Among the most popular such implementations are long multipliers, Booth multipliers and Wallace-tree multipliers. In this work, we focus only on long multipliers and Wallace-tree multipliers. The study of our heuristic pre-processing step for systems containing Booth multipliers is left as part of future work.

Long multiplier: Consider bit-vectors x and y that are partitioned into k blocks of width w bits each. Thus the total width of each bit-vector is $k \cdot w$. The long multiplier decomposes the multiplication of two $(k \cdot w)$ -bit wide bit-vectors into k^2 multiplications of w -bit wide bit-vectors. The corresponding k^2 products, called *partial products*, are then added with appropriate left-shifts to obtain the final result. The following notation is typically used to illustrate long multiplication.

$$\begin{array}{rcccc}
 & x_k & \dots & x_1 & \\
 & y_k & \dots & y_1 & * \\
 \hline
 & x_k * y_1 & \dots & x_1 * y_1 & \\
 & \cdot & \vdots & \cdot & \\
 x_k * y_k & \dots & x_1 * y_k & & +
 \end{array}$$

Here, the $x_i * y_j$ s are the partial products. The partial product $x_i * y_j$ is left shifted $(i + j - 2) \cdot w$ bits before being added. In the above representation, all partial products that are left-shifted by the same amount are aligned in a single column. After the left shifts, all the partial results are added in some order. Note that the bit-width of each partial product is $2 \cdot w$. Since the syntax of QF_BV requires the bit-widths of the arguments and result of the $*$ operator to be the same, we denote the partial product $x_i * y_j$ as $(0^w \bullet x_i) * (0^w \bullet y_j)$ for our purposes. Note further that the bits of the partial products in neighbouring columns (in the above representation of long multiplication) overlap; hence the sums of the various columns can not be simply concatenated. The long multiplication algorithm does not specify the order of addition of the shifted partial products. Therefore, there are several possible implementations for a given k and w .

Example 1. Consider bit-vectors v_3, v_2, v_1, u_3, u_2 , and u_1 , each of bit-width 2. Let us apply long multiplication on $v_3 \bullet v_2 \bullet v_1$ and $u_3 \bullet u_2 \bullet u_1$. We obtain the following partial products.

$$\begin{array}{rcccc}
 & v_3 & v_2 & v_1 & \\
 & u_3 & u_2 & u_1 & * \\
 \hline
 & v_3 * u_1 & v_2 * u_1 & v_1 * u_1 & \\
 & v_3 * u_2 & v_2 * u_2 & v_1 * u_2 & \\
 v_3 * u_3 & v_2 * u_3 & v_1 * u_3 & & +
 \end{array}$$

The following term is one (of several) possible combinations of the partial products using concatenations and summations to obtain the final product.

$$((v_3 * u_3) \bullet (v_3 * u_1) \bullet (v_1 * u_1)) + (0^2 \bullet (v_2 * u_3) \bullet (v_2 * u_1) \bullet 0^2) + (0^2 \bullet (v_3 * u_2) \bullet (v_1 * u_2) \bullet 0^2) + (0^4 \bullet (v_2 * u_2) \bullet 0^4) + (0^4 \bullet (v_1 * u_3) \bullet 0^4)$$

Note that we did not concatenate two partial products that appear next to each other in the tabular representation, because their bits can potentially overlap.

Example 2. Consider bit-vectors v_1, v_2, u_1 , and u_2 , each of bit-width 2. Let us apply long multiplication on $v_2 \bullet 0^2 \bullet v_1$ and $u_2 \bullet v_2 \bullet u_1$. We obtain the following partial products.

		v_2	0^2	v_1	
		u_2	v_2	u_1	$*$
		$v_2 * u_1$	0^4	$v_1 * u_1$	
	$v_2 * v_2$	0^4	$v_1 * v_2$		
$v_2 * u_2$	0^4	$v_1 * u_2$		$+$	

Note that while adding the shifted partial products, if the non-zero bits of a subset of shifted partial products do not overlap, then we can simply concatenate them to obtain their sum. Finally, we can sum the concatenated vectors thus obtained to calculate the overall product. The following is one possible combination of concatenations and summations for the long multiplication in this example.

$$(0^4 \bullet (v_1 * u_2) \bullet (v_1 * u_1)) + ((v_2 * u_2) \bullet (v_2 * u_1) \bullet 0^4) + (0^2 \bullet (v_2 * v_2) \bullet (v_1 * v_2) \bullet 0^2)$$

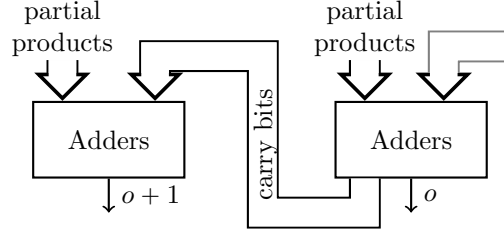
Example 3. As another interesting example, consider long multiplication applied to $v_2 \bullet 0^2 \bullet v_2$ and $0^2 \bullet v_1 \bullet v_1$, where v_1 and v_2 have bit-width 2. We obtain the following partial products.

	v_2	0^2	v_2	
	0^2	v_1	v_1	$*$
		$v_1 * v_2$	0^4	$v_1 * v_2$
$v_1 * v_2$	0^4	$v_1 * v_2$	$+$	

Note that, if we had applied long multiplication to $v_1 \bullet 0^2 \bullet v_1$ and $0^2 \bullet v_2 \bullet v_2$, we would have obtained the same set of shifted partial products. This shows that simply knowing the collections of shifted partial products does not permit uniquely determining the multiplier and multiplicand. Recall that this dilemma was alluded to in Section 1, when discussing pattern-matching based re-writing.

Wallace tree multiplier[16]: A Wallace tree decomposes the multiplication of two bit-vectors all the way down to single bits. Let us consider bit-vectors x and y that are accessed in blocks of size 1 bit and are of bit-width k . In a Wallace tree, a partial product $x_i * y_j$ is the multiplication of single bits, and hence is implemented as the conjunction of the bits, i.e., $x_i \wedge y_j$. There is no carry generated due to the multiplication of single bits. The partial product $x_i * y_j$ is aligned with the $(i + j - 2)$ th bit of output. Let us consider the o th output bit.

All the partial products that are aligned to o are summed using full adders and half adders. Specifically, full adders are used if more than two bits remain to be summed, while half adders are used if only two bits remain to be summed. The carry bits that are generated by adding the partial products for the o th output bit are aligned to the $(o + 1)$ th output bit. Finally, these carry bits are added to the partial products generated for $(o + 1)$ th bit using adders, as illustrated in the following figure.



From the above discussion, it is clear that neither a long multiplier nor a Wallace tree multiplier completely specifies a multiplier implementation. Therefore, there are several ways to implement a multiplier, and it is non-trivial to verify that an implementation is correct.

3 Pattern detection

In this section, we present algorithms that attempt to match multiplications that are decomposed using long or Wallace tree multiplication. If we match some terms of the input formula as instances of multiplication, we add tautologies stating that the terms are equivalent to the product of the matched bit-vectors. Our matching method may find multiple matches for a term. We add a tautology for each match to the input formula, and solve using an available solver.

3.1 Matching long multiplication

In Algorithm 1, we present a function `MATCHLONG` that takes a `QF_BV` term t and returns a set of matched multiplications. This algorithm and the subsequent ones are written such that as soon as it becomes clear that no multiplication can be matched, they return the empty set. At line 1 of Algorithm 1, we match t with a sum of concatenations, and if the match fails then we conclude that t does not encode a long multiplication. At line 2, we find a partial product among s_{ij} and extract the block size w used by the long multiplication. The loop at line 4 populates the vector of the set of partial products A . A_i denotes the partial products that are aligned at the i th block. Each s_{ij} must either be 0 or a partial product of the form mentioned in the condition at line 7. Otherwise, t is declared unmatched at line 9. At line 5, we compute the alignment o for s_{ij} . If s_{ij} happens to be a non-zero partial product, it is inserted in A_o at line 8. At line 10, we call `GETMULTOPERANDS` to identify the operands of the long multiplication from A if t indeed encodes a long multiplication.

Algorithm 1 MATCHLONG(t)

Require: t : a term in $\mathbf{QF_BV}$

Ensure: M : matched multiplications $:= \emptyset$

```
1: if  $t == (s_{1k_1} \bullet \dots \bullet s_{1l_1}) + \dots + (s_{pk_p} \bullet \dots \bullet s_{pl_1})$  then
2:   Let  $w$  be such that some  $s_{ij}$  is of the form  $(0^w \bullet a) * (0^w \bullet b)$ , we have  $\text{len}(a) = \text{len}(b) = w$ 
3:    $\Lambda := \lambda i. \emptyset$ 
4:   for each  $s_{ij}$  do
5:      $o := (\sum_{j' < j} \text{len}(s_{ij'})) / w + 1$ 
6:     if  $s_{ij} == 0$  then continue;
7:     if  $s_{ij} == (0^w \bullet a) * (0^w \bullet b)$  and  $\text{len}(a) = \text{len}(b) = w$  then
8:        $\Lambda_o.\text{insert}(a * b)$ 
9:     else return  $\emptyset$ 
10:  return GETMULTOPERANDS( $\Lambda, w$ )
11: return  $\emptyset$ 
```

3.2 Partial products to operands

In Algorithm 2, we present a function GETMULTOPERANDS that takes a vector of multiset of partial products Λ and a block-width w , and returns a set of matched multiplications. The algorithm proceeds by incrementally choosing a pair of operands with insufficient information and backtracks if the guess is found to be wrong.

At line 1, we compute h and l that establishes the range of the search for the operands. We maintain two candidate operands x and y of bit-width $h.w$. We also maintain a vector of bits *backtrack* that encodes the possibility of flipping the uncertain decisions. Due to the scheme of the long multiplication, the highest non-empty entry in Λ must be a singleton set. If Λ_h contains a single partial product $a * b$, we assign x_h and y_h the operands of $a * b$ arbitrarily. We assign **ff** to *backtrack_h*, which states that there is no need of backtracking at index h . If Λ_h does not contain a single partial product, we declare the match has failed by returning \emptyset . The loop at line 8 iterates over index i from h to 1. In each iteration, it assigns values to x_i , y_i , and *backtrack_i*.

The algorithm may not have enough information at the i th iteration and the chosen value for x_i and y_i may be wrong. Whenever, the algorithm realizes that such a mistake has happened it jumps to line 31. It increases back the value of i to the latest i' that allows backtracking. It swaps the assigned values of x_i and y_i , and disables future backtracking to i by setting *backtrack_i* to **ff**.

Let us look at the loop at line 8 again. We also have variables l_x and l_y that contain the least index of the non-zero blocks in x and y , respectively. At line 9, we decrement i and Λ_i is copied to C . At index i , the sum of the aligned partial products is the following.

$$x_h * y_i + \underbrace{x_{h-1} * y_{i+1} + \dots + x_{i+1} * y_{h-1}}_{\text{operands seen at the earlier iterations}} + x_i * y_h$$

Algorithm 2 GETMULTOPERANDS(Λ, w)

Require: Λ : array of multisets of the partial products

Ensure: M : matched multiplications $:= \emptyset$

```
1: Let  $l$  and  $h$  be the smallest and largest  $i$  such that  $\Lambda_i \neq \emptyset$ , respectively.
2:  $x, y$  : candidate operands of bit-width  $h.w$  that are accessed in blocks of size  $w$ 
3: if  $\Lambda_h == \{a * b\}$  then
4:    $x_h := a; y_h := b; backtrack_h := \mathbf{ff};$ 
5: else
6:   return  $\emptyset$ 
7:  $i := h; l_x := h; l_y = h;$ 
8: while  $i > 1$  do
9:    $i := i - 1; C := \Lambda_i$ 
10:  for  $j \in (h - 1)..(i + 1)$  do
11:    if  $x_j \neq 0$  and  $y_{h+i-j} \neq 0$  then
12:      if  $x_j * y_{h+i-j} \notin C$  then goto BACKTRACK
13:       $C := C - \{x_j * y_{h+i-j}\}$ 
14:  match  $C$  with
15:     $\{x_h * b, y_h * d\} \rightarrow x_i := d; y_i := b; backtrack_i := (x_h = y_h);$ 
16:     $\{x_h * y_h\} \rightarrow x_i := 0; y_i := x_h; backtrack_i := \mathbf{tt};$ 
17:     $\{x_h * b\} \rightarrow x_i := 0; y_i := b; backtrack_i := (x_h = y_h);$ 
18:     $\{y_h * b\} \rightarrow x_i := b; y_i := 0; backtrack_i := \mathbf{ff};$ 
19:     $\{\}$   $\rightarrow x_i := 0; y_i := 0; backtrack_i := \mathbf{ff};$ 
20:     $\_ \rightarrow$  goto BACKTRACK;
21:  if  $x_i \neq 0$  then  $l_x := i$ 
22:  if  $y_i \neq 0$  then  $l_y := i$ 
23:  if  $l_x + l_y - h < 1$  then goto BACKTRACK;
24:  if  $i = 1$  then
25:    for  $o \in 0..(l - 1)$  do
26:       $x' :=$  Right shift  $x$  until  $o$  trailing 0 blocks in  $x$ 
27:       $y' :=$  Right shift  $y$  until  $l - 1 - o$  trailing 0 blocks in  $y$ 
28:       $M := M \cup \{x' * y'\}$ 
29:  else
30:    continue;
31:  BACKTRACK:
32:    Choose smallest  $i' \in h..(i + 1)$  such that  $backtrack_{i'} = \mathbf{tt}$ 
33:    if no  $i'$  found then return  $M$ 
34:     $i := i'; \text{SWAP}(x_i, y_i); backtrack_i := \mathbf{ff}$ 
```

We have already chosen the operands of the middle partial products in the previous iterations. Only the partial products at the extreme ends have y_i and x_i that are not assigned yet. In the loop at line 10, we remove the middle partial products. If any of the needed partial product is missing then we may have made a mistake earlier and we jump for backtracking. After the loop, we should be left with at most two partial products in C corresponding to $x_h * y_i$ and $x_i * y_h$. We match C with the five patterns at lines 14-19 and update x_i , y_i , and $backtrack_i$ accordingly. If none of the patterns match, we jump for backtracking at line 20. In some cases we clearly determine the value of x_i and y_i , and we are not certain

Algorithm 3 MATCHWALLACETREE(t)

Ensure: t : a term in QF_BV

```
1: if  $t == (t_k \bullet \dots \bullet t_1)$  then
2:    $\Lambda := \lambda i. \emptyset$ ;  $\Delta$  : vector of multiset of terms :=  $\lambda i. \emptyset$ 
3:   for  $i \in 1..k$  do
4:     if  $\text{len}(t_i) \neq 1$  then return  $\emptyset$ 
5:      $S := \{t_i\}$ ;  $\Delta_i := \{t_i\}$ 
6:     while  $S \neq \emptyset$  do
7:        $t \in S$ ;  $S := S - \{t\}$ 
8:       if  $t == s_1 \oplus \dots \oplus s_p$  then
9:          $S := S \cup \{s_1, \dots, s_p\}$ ;  $\Delta_i := \Delta_i \cup \{s_1, \dots, s_p\}$ 
10:      else if  $t == \text{carryFull}(a, b, c)$  and  $a, b, c, a \oplus b, a \oplus b \oplus c \in \Delta_{i-1}$  then
11:         $\Delta_{i-1} := \Delta_{i-1} - \{a, b, c, a \oplus b\}$ 
12:      else if  $t == \text{carryHalf}(a, b)$  and  $a, b, a \oplus b \in \Delta_{i-1}$  then
13:         $\Delta_{i-1} := \Delta_{i-1} - \{a, b\}$ 
14:      else if  $t == a \wedge b$  then
15:         $\Lambda_i.\text{insert}(a * b)$ ;
16:      else return  $\emptyset$ 
17:      if  $\Delta_{i-1} \neq \{t_{i-1}\}$  then return  $\emptyset$ 
18:      return GETMULTOPERANDS( $\Lambda, 1$ )
19: return  $\emptyset$ 
```

in the other cases. We set backtrack_i to **tt** in the uncertain cases to indicate that we may return back to index i and swap x_i and y_i . In the following list, we discuss the uncertain cases.

line 15: If C has two elements $x_h * b$ and $y_h * d$, there is an ambiguity in choosing x_i and y_i if $x_h = y_h$.

line 16: If C has a single element $x_h * y_h$, there are two possibilities.

line 17: If $C = \{x_h * b\}$ and b is not y_h then similar to the first case there is an ambiguity in choosing x_i and y_i if $x_h = y_h$. Line 18 is similar.

line 19: If C is empty then there is no uncertainty.

At line 21-22, we update l_x and l_y appropriately. The condition at line 23 ensures that the expected least index i such that $\Lambda_i \neq \emptyset$ is greater than 0. At line 24, we check if $i = 1$, which means a match has been successful. x and y are not the operands that we are seeking. They are aligned to the left boundary of Λ , which allows us to use an uniform indexing scheme in the algorithm. To find the appropriate operands, we need to right shift x and y such that the total number of their trailing zero blocks is $l - 1$. We add the matched $x' * y'$ to the match store M . And, the algorithm proceeds for backtracking to find if more matchings exist.

3.3 Matching Wallace tree multiplication

A Wallace tree has a cascade of adders that take partial products and carry bits as input to produce the output bits. In our matching algorithm, we find the set

Algorithm 4 OURSOLVER(F)

Require: F : a QF_BV formula

Ensure: sat/unsat/undef

```
1: SMTSOLVER.add( $F$ )
2: for each subterm  $t$  in  $F$  do
3:   if  $M := \text{MATCHLONG}(t) \cup \text{MATCHWALLACETREE}(t)$  then
4:     for each  $x * y \in M$  do
5:       SMTSOLVER.add( $[x * y = t]$ )
6: return SMTSOLVER.checkSat()
```

of inputs to the adders for an output bit and classify them into partial products and carry bits. The half and full adders are defined as follows.

$$\begin{aligned} \text{sumHalf}(a, b) &= a \oplus b & \text{sumFull}(a, b, c) &= a \oplus b \oplus c \\ \text{carryHalf}(a, b) &= a \wedge b & \text{carryFull}(a, b, c) &= (a \wedge b) \vee (b \wedge c) \vee (c \wedge a) \end{aligned}$$

The sum outputs of half/full adders are the results of xor operations of inputs. To find the input to the cascaded adders, we start from an output bit and follow backward until we find the input that is not the result of some xor.

In Algorithm 3, we present a function MATCHWALLACE that takes a QF_BV term t and returns a set of matched multiplications. At line 1, t is matched with a concatenation of single bit terms t_k, \dots, t_1 . Similar to Algorithm 1, we maintain the partial product store Λ . For each i , we also maintain the multiset of terms Δ_i that were used as inputs to the adders for the i th bit. In the loop at line 6, we traverse down the subterms until a subterm is not the result of a xor. In the traversal, we also collect the inputs of the visited xors in Δ_i , which will help us in checking that all the carry inputs in adders for t_{i+1} are generated by the adders for t_i . If the term t is not the result of xors then we have the following possibilities.

line 10-13: If t is the carry bit of a half/full adder, and the inputs, the intermediate result of the sum bit, and the output sum bit of the adder are in Δ_{i-1} then we remove the inputs and intermediate result of the adder from Δ_{i-1} .

We do not remove the output sum bit from Δ_{i-1} , since it may be used as input to some other adder.

line 14-15: If t is a partial product, we record it in Λ_i .

line 16: Otherwise, we return \emptyset .

At line 17, we check that $\Delta_{i-1} = \{t_{i-1}\}$, i.e., all carry bits from the adders for t_{i-1} are consumed by the adders for t_i exactly once. Again if the check fails, we return \emptyset . After the loop at line 3, we have collected the partial products in Λ . At line 18, we call GETMULTOPERANDS($\Lambda, 1$) to get all the matching multiplications.

3.4 Our solver

Using the above pattern matching algorithms, we modify an existing solver, generically called SMTSOLVER; as presented in Algorithm 4. OURSOLVER adds

the input formula F in SMTSOLVER. For every subterm of F , we attempt to match with both long multiplication or Wallace tree multiplication. For each discovered matching $x * y$, we add a bit-vector tautology $[x * y = t]$ to the solver, which is obtained after appropriately zero-padding x , y , and t .

4 Correctness

We need to prove that each $[x * y = t]$ added in OURSOLVER is a tautology. First we will prove the correctness of GETMULTOPERANDS. If either of x or y is zero, we assume the term $x * y$ is also simplified to zero.

Theorem 1 *If $x * y \in \text{GETMULTOPERANDS}(\Lambda, w)$, then*

$$\Lambda_i = \{x_1 * y_i, \dots, x_i * y_1\}, \text{ and if } x_j * x_k \text{ is non-zero, } j + k \leq h,$$

where x_ℓ and y_ℓ are the ℓ th block of x and y of size w , respectively.

Proof. After each iteration of the loop at line 8, if no backtracking is triggered, the loop body ensures that the following holds, which one may easily check.

$$\Lambda_i = \{x_h * y_i, x_{h-1} * y_{i+1}, \dots, x_i * y_h\} \quad (1)$$

Due to the above equation, if $x_j * y_k \in \Lambda_i$, $i = j + k - h$. If the program enters at line 25, it has a successful match and $i = 1$. Since $l_x + l_y - h \geq 1$, $\Lambda_l = \{x_{l_x} * y_{l_y}\}$ and $l = l_x + l_y - h$. We choose $o \leq l$, and shift x and y according to lines 26-27. After the shift, we need to write equation (1) as follows.

$$\Lambda_i = \{x_{h-(l_x-o-1)} * y_{i-(l_y-l+o)}, \dots, x_{i-(l_x-o-1)} * y_{h-(l_y-l+o)}\}. \quad (2)$$

We can easily verify that the sum of the indexes in each of the partial products is $i+1$. Since all x_k is zero for $k > h - (l_x - o)$ and all y_k is zero for $k > h - (l_y - l + o)$, we may rewrite equation (2) as follows.

$$\Lambda_i = \{x_1 * y_i, \dots, x_i * y_1\}.$$

Since the largest non-zero blocks in x and y are $h - (l_x - o - 1)$ and $h - (l_y - l + o)$, all the non-zero partial products appear at index less than or equal to h in Λ .

Theorem 2 *If $m * n \in \text{MATCHLONG}(t)$, $[m * n = t]$ is a tautology.*

Proof. We collect partial products with appropriate offsets o at line 5. The pattern of t indicates that the net result is the sum of the partial products with the respective offsets. GETMULTOPERANDS(Λ, w) returns the matches that produces the sums. Therefore, $[m * n = t]$ is a tautology.

Theorem 3 *If $m * n \in \text{MATCHWALLACETREE}(t)$, $[m * n = t]$ is a tautology.*

Proof. All we need to show is that t sums the partial products stored in Δ . The rest of the proof follows the previous theorem.

Each bit t_i must be the sum of the partial products Δ_i and the carry bits produced by the sum for t_{i-1} . The algorithm identifies the terms that are added to obtain t_i and collects the intermediate results of the sum in Δ_i . We only need to prove that the terms that are not identified as partial products are carry bits of the sum for t_{i-1} . Let us consider such a term t . Let us suppose the algorithm identifies t as an output of the carry bit circuit of a full adder (half adder case is similar) with inputs a , b , and c . The algorithm also checks that a , b , c , $a \oplus b$ and $a \oplus b \oplus c$ are the intermediate results of the sum for t_{i-1} . Therefore, t is one of the carry bits. Since a , b , $a \oplus b$ and c are removed from Δ_{i-1} after the match of the adder, all the identified adders are disjoint. Since we require that all the elements of Δ_{i-1} are eventually removed except t_{i-1} , all carry bits are added to obtain t_i . Therefore, Δ has the expected partial products of a Wallace tree.

5 Experiments

We have implemented³ our algorithms as a part of the Z3 SMT solver. We evaluate the performance of our algorithms using benchmarks that are derived from hardware verification problems. We compare our tool with Z3, BOOLECTOR and CVC4. Our experiments show that while the solvers time out on most of the benchmarks without our heuristic, a portfolio solver using our heuristic produces results within the prescribed time limit.

Implementation We have added about 1500 lines of code in the bit vector rewrite module of Z3 because it allows easy access to the abstract syntax tree of the input formula. We call this version of Z3 as instrumented-Z3. An important aspect of the implementation is its ability to exit early if the match is going to fail. We implemented various preliminary checks including the ones mentioned in Algorithm 1. For example, we ensure that the size of Δ_i is upper bounded appropriately as per the scheme of long multiplication. We exit as soon as the upper bound is violated. We have implemented three versions of OURSOLVER by varying the choice of SMTSOLVER. We used Z3, BOOLECTOR, and CVC4 for the variations.

In each case we stop the instrumented-Z3 solver after running our matching algorithms, print the learned tautologies in a file along with the input formula, and run the solvers in a separate process on the pre-processed formula. The time taken to run our matching algorithms and generate the pre-processed formula is less than one second across all our benchmarks, and hence is not reported. We also experimented by running instrumented-Z3 standalone and found the run times to be similar to that of Z3 running on the pre-processed formula; hence the run times for instrumented-Z3 are not reported. We use the following versions of the solvers: Z3(4.4.2), BOOLECTOR(2.2.0), CVC4(1.4) for our experiments.

³ <https://github.com/rahuljain1989/Bit-vector-multiplication-pattern>

Benchmarks Our experiments were run on 20 benchmarks. Initially, we considered the motivating example described in Section 1 involving long multiplication that was not solved by any of the solvers in 24 hours. This example inspired our current work and to evaluate it we generated several similar benchmarks. For long multiplication, we generated benchmarks by varying three characteristics. Firstly the total bit-width of the input bit-vectors, secondly the width of each block, and thirdly assigning specific blocks as equal or setting them to zero. Our benchmarks were written in SYSTEMVERILOG and fed to STEWord [13], a hardware verification tool. STEWord takes a SYSTEMVERILOG design and a specification of how it is to be driven as input, and generates an SMT formula in SMT1 format. We convert the SMT1 formula to SMT2 format using BOOLECTOR. In the process, BOOLECTOR extensively simplifies the input formula but retains the overall structure. We have generated benchmarks also for Wallace tree multiplier similar to the long multiplication. For n -bit Wallace tree multiplier, we have written a script that takes n as input and generates all the files needed as input by STEWord. All our benchmarks correspond to the system implementation satisfying the specified property: in other words, the generated SMT formulas were unsatisfiable. For satisfiable formulas the solver was able to find satisfying assignments relatively quickly, both with and without our heuristic. Hence, we do not report results on satisfiable formulas.

Results We compare our tool with Z3, BOOLECTOR and CVC4. In Tables 1-2, we present the results of the experiments. We chose timeout to be 3600 seconds. In Table 1, we present the timings of the long multiplication and Wallace tree multiplier experiments. The first 13 rows correspond to the long multiplication experiments. The columns under SMTSOLVER are the run times of the solvers to prove the unsatisfiability of the input benchmark. The solvers timed out on most of the benchmarks.

The next three columns present the run times of the three versions of OURSOLVER to prove the satisfiability of the benchmarks. OURSOLVER with CVC4 makes best use of the added tautologies. CVC4 is quickly able to infer that the input formula and the added tautologies are negations of each other justifying the timings. OURSOLVER with BOOLECTOR and Z3 does not make the above inference, leading to more running times. BOOLECTOR and Z3 bit blast the benchmarks having not been able to detect the structural similarity. However, the added tautologies help BOOLECTOR and Z3 to reduce the search space, after the SAT solver is invoked on the bit-blasted formulas.

The last 7 rows correspond to the Wallace tree multiplier experiments. Since the multiplier involves a series of half and full adders, the size of the input formula increases rapidly as the bit vector width increases. Despite the blowup in the formula size, OURSOLVER with Z3 is quickly able to infer that the input formula and the added tautology are negations of each other. However, OURSOLVER with BOOLECTOR and CVC4 do not make the inference, leading to larger run times. This is because of the syntactic structure of the learned tautology from our implementation inside Z3. The input formula has ‘and’ and ‘not’ gates as its building blocks, whereas Z3 transforms all ‘ands’ to ‘ors’. Therefore, the added

Table 1: Multiplication experiments. Times are in seconds. PORTFOLIO column is the least timing among the solvers. Bold entries are the minimum time.

Benchmark	SMTSOLVER			OURSOLVER			PORTFOLIO
	Z3	BOOLECTOR	CVC4	Z3	BOOLECTOR	CVC4	
base	184.3	42.2	16.54	0.53	43.5	0.01	0.01
ex1	2.99	0.7	0.36	0.33	0.8	0.01	0.01
ex1_sc	t/o	t/o	t/o	1.75	t/o	0.01	0.01
ex2	0.78	0.2	0.08	0.44	0.3	0.01	0.01
ex2_sc	t/o	1718	2826	3.15	1519	0.01	0.01
ex3	1.38	0.3	0.08	0.46	0.7	0.01	0.01
ex3_sc	t/o	1068	t/o	3.45	313.2	0.01	0.01
ex4	0.46	0.2	0.03	0.82	0.2	0.01	0.01
ex4_sc	287.3	62.8	42.36	303.6	12.8	0.01	0.01
sv_assy	t/o	t/o	t/o	0.07	t/o	0.01	0.01
mot_base	t/o	t/o	t/o	13.03	1005	0.01	0.01
mot_ex1	t/o	t/o	t/o	1581	13.8	0.01	0.01
mot_ex2	t/o	t/o	t/o	2231	13.7	0.01	0.01
wal_4bit	0.09	0.05	0.02	0.09	0.1	0.04	0.02
wal_6bit	2.86	0.6	0.85	0.28	0.8	14.36	0.28
wal_8bit	209.8	54.6	225.1	0.59	30.0	3471	0.59
wal_10bit	t/o	1523	t/o	1.03	98.6	t/o	1.03
wal_12bit	t/o	t/o	t/o	1.55	182.3	t/o	1.55
wal_14bit	t/o	t/o	t/o	2.27	228.5	t/o	2.27
wal_16bit	t/o	t/o	t/o	2.95	481.7	t/o	2.95

tautology has no ‘ands’. The difference in the syntactic structure between the input formula and the added tautology makes it difficult for BOOLECTOR and CVC4 to make the above inference.

We have seen that the solvers sometimes fail to apply word level reasoning even after adding the tautologies. In such cases, the solvers bit blast the formula and run a SAT solver. In Table 2, we present the number of conflicts and decisions within the SAT solvers. The number of conflicts and decisions on running OURSOLVER with the three solvers, are considerably less than their SMTSOLVER counterparts in most of the cases. This demonstrates that the tautologies also help in reducing the search inside the SAT solvers. OURSOLVER with CVC4 has zero conflicts and decisions for all the long multiplication experiments, because the word level reasoning solved the benchmarks. Similarly, OURSOLVER with Z3 has zero conflicts and decisions for all the Wallace tree multiplier experiments.

Limitations Although our initial results are promising, our current implementation has several limitations as well. We have only considered a limited space of low-level multiplier representations. Actual representations may include several other optimizations, e.g., multiplying with constants using bit-shifting etc. Multiplier operations may also be applied recursively, e.g., the partial products of a long multiplication may be obtained using Wallace tree multiplier. While we have noticed significant benefits of adding tautological assertions encoding the

Table 2: Conflicts and decisions in the experiments. M stands for millions. k stands for thousands.

Benchmark	SMT Solver						Our Solver					
	Z3		BooleX		CVC4		Z3		BooleX		CVC4	
	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions
base	172k	203k	170k	228k	127k	148k	724	1433	148k	194k	0	0
ex1	7444	9065	7320	9892	8396	10k	474	890	7090	9558	0	0
ex1.sc	t/o —	t/o —	t/o 5.6M	t/o 7.7M	t/o 2.1M	t/o 2.3M	2564	5803	t/o 5M	t/o 6.8M	0	0
ex2	2067	2599	1789	2612	2360	3374	919	1420	1747	2526	0	0
ex2.sc	t/o —	t/o —	3.3M	4.9M	1.9M	2.3M	5076	8981	2.7M	4.3M	0	0
ex3	4109	5402	1682	3166	3374	4754	905	1321	3882	7305	0	0
ex3.sc	t/o —	t/o —	3.8M	5.9M	t/o 2.9M	t/o 3.6M	4814	9012	805k	1.4M	0	0
ex4	647	801	612	715	463	588	630	918	405	519	0	0
ex4.sc	143k	165k	130k	165k	110k	130k	115k	138k	67k	114k	0	0
sv.assy	t/o —	t/o —	t/o 5.3M	t/o 9.8M	t/o 1.8M	t/o 2.5M	0	0	t/o 4.7M	t/o 9.4M	0	0
mot-base	t/o —	t/o —	t/o 6M	t/o 10M	t/o 2.2M	t/o 2.9M	12k	30k	2.4M	5.5M	0	0
mot-ex1	t/o —	t/o —	t/o 4.4M	t/o 6.1M	t/o 1.7M	t/o 2M	280k	409k	30k	57k	0	0
mot-ex2	t/o —	t/o —	4.5M	6.3M	t/o 1.7M	t/o 1.9M	358k	496k	30k	57k	0	0
wal.4bit	363	435	283	343	396	479	0	0	300	378	442	486
wal.6bit	8077	9831	6887	9544	11k	12k	0	0	8523	12k	68k	54k
wal.8bit	180k	209k	177k	249k	1.2M	1.1M	0	0	94k	174k	5.9M	3.8M
wal.10bit	t/o —	t/o —	2.7M	3.7M	t/o 5.4M	t/o 2.2M	0	0	249k	519k	t/o 2.8M	t/o 1.2M
wal.12bit	t/o —	t/o —	t/o 5.2M	t/o 6M	t/o 4.1M	t/o 1.9M	0	0	416k	855k	t/o 2.3M	t/o 916k
wal.14bit	t/o —	t/o —	t/o 4.9M	t/o 6.5M	t/o 3M	t/o 907k	0	0	500k	999k	t/o 1.2M	t/o 412k
wal.16bit	t/o —	t/o —	t/o 4.8M	t/o 6.6M	t/o 1.9M	t/o 512k	0	0	941k	2M	t/o 672k	t/o 196k

equivalence of pattern-matched terms with bit-vector multiplication, in general, adding such assertions can hurt solving time as well. This can happen if, for example, the assertions are themselves bit-blasted by the solver, thereby overwhelming the underlying SAT solver. In addition, the added assertions may be re-written by optimization passes of the solver, in which case they may not help in identifying sub-terms representing multiplication in the overall formula. Since the nature of our method is to exploit the potential structure in the input, we must also adapt all parts of the solver to be aware of the sought structure as part of our future work. We are currently working to tag the added assertions such that they are neither simplified in pre-processing nor bit-blasted by the solver. Instead, they should only contribute to the word-level reasoning. Note that our current benchmarks are also limited in the sense that they do not include examples where multiplication is embedded deep in a large formula. We are working to make our implementation robust such that it can reliably work on larger examples, in particular on all the SMT-LIB benchmarks. More results in this direction may be found at [28].

6 Related Work

The quest for heuristic strategies for improving performance of SMT solvers dates back to the early days of SMT solving. An excellent exposition on several important early strategies can be found in [11]. The importance of orchestrating different heuristics in a problem-specific manner has also been highlighted in [12]. The works that come closest to our work are those developed in the context of verifying hardware implementations of word-level arithmetic operations. There is a long history of heuristics for identifying bit-vector (or word-level) operators from gate-level implementations (see, for example, [19,20,22,23] for a small sampling). The use of canonical representations of arithmetic operations have also been explored in the context of verifying arithmetic circuits like multipliers (see [29,30], among others). However, these representations usually scale poorly with the bit-width of the multiplier. Equivalence checkers determine if two circuits, possibly designed in different ways, implement the same overall functionality. State-of-the-art hardware equivalence checking tools, like Hector [31], make use of sophisticated heuristics like structural similarities between sub-circuits, complex rewrite rules and heuristic sequencing of reasoning engines to detect equivalences between two versions of a circuit. Since these efforts are primarily targeted at establishing the functional equivalence of one circuit with another, replacing one circuit configuration with another often works profitably. However, as argued in Section 1, this is not always desirable when checking the satisfiability of a formula obtained from word-level BMC or STE. Hence, our approach differs from the use of rewrites used in hardware equivalence checkers, although there are close parallels between the two.

It is interesting to note that alternative representations of arithmetic operators are internally used in SMT solvers when bit-blasting high-level arithmetic operators. For example, Z3 [17] uses a specific Wallace-tree implementation of

multiplication when blasting multiplication operations. Since a wide multiplication operator admits multiple Wallace-tree implementation, this may not match terms encoding the Wallace-tree implementation of the same operator in another part of the formula. Similar heuristics for bit-blasting arithmetic operators are also used in other solvers like BOOLECTOR [2] and CVC4 [18]. However, none of these are intended to help improve the performance of the solver. Instead, they are used to shift the granularity of reasoning from word-level to bit-level for the sake of completeness, but often at the price of performance.

7 Conclusion and future work

We have shown how adding tautological assertions that assert the equivalence of different representations of bit-vector multiplication can significantly improve the performance of SMT solvers. We are currently extending our procedure to support Booth multiplier and other more complex arithmetic patterns. We are also working to add proof generation support for the added tautological assertions. We could not include proof generation in this work, since the basic infrastructure of proof generation is missing in Z3 bit-vector rewriter module.

References

1. Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
2. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings*, pages 174–177, 2009.
3. EBMC. <http://www.cprover.org/ebmc/>.
4. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.
5. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings*, pages 427–443, 2012.
6. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1–4, 2005, Revised Lectures*, pages 364–387, 2005.
7. Y. Naveh and R. Emek. Random stimuli generation for functional hardware verification as a cp application. In *Proc. of CP*, pages 882–882, 2005.
8. Y. Naveh, M. Rimón, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. Constraint-based random stimuli generation for hardware verification. In *Proc. of AAAI*, pages 1720–1727, 2006.

9. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
10. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.
11. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
12. Leonardo de Moura and Grant Olney Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
13. Supratik Chakraborty, Zurab Khasidashvili, Carl-Johan H. Seger, Rajkumar Gajavelly, Tanmay Haldankar, Dinesh Chhatani, and Rakesh Mistry. Word-level symbolic trajectory evaluation. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 128–143, 2015.
14. Long multiplication. https://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication.
15. Booth’s multiplication algorithm. https://en.wikipedia.org/wiki/Booth's_multiplication_algorithm.
16. Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.
17. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.
18. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.
19. Dominik Stoffel and Wolfgang Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(5):586–597, 2004.
20. Cunxi Yu and Maciej J. Ciesielski. Automatic word-level abstraction of datapath. In *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016*, pages 1718–1721, 2016.
21. Alfred Kölbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 196–201, 2009.
22. Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. Reverse engineering digital circuits using functional analysis. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1277–1280, 2013.
23. Wenchao Li, Adria Gascón, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 67–74, 2013.

24. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.
25. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
26. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
27. Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 203–208, 1997.
28. Supratik Chakraborty, Ashutosh Gupta, and Rahul Jain. Matching multiplications in bit-vector formulas. <https://www.tcs.tifr.res.in/~agupta/papers/vmcai17-extended.pdf>.
29. Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541, 1995.
30. Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1048–1053, 2016.
31. HECTOR. <http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/hector.aspx>.